Gabriel Morales

CSC 258 – Parallel & Distributed Systems (SPRING 2017)

Semester Project – Hamiltonian Path Problem Solver for Grid Graphs

WRITEUP

How to compile:

If you have Windows and a version of Visual Studio installed on your computer, you compile using a makefile by first adding the "nmake" binary path (packaged with VS) into your path and running:

```
$nmake /f windows\Makefile.mak
```

For Unix and Linux based systems, you can compile the makefile as follows:

```
$make -f unix-linux/Makefile
```

How to run:

On Windows:

```
$windows\run.bat [args]
```

On Unix/Linux:

```
$sh unix-linux/run.sh [args]
```

The arguments to the script are passed as written to the java program which parses the them to perform specific behavior. The following are the options available for the program:

```
Usage: java Driver [-options]

OPTIONS

    -a, --args <rows> <columns>     create a board with the specified dimensions

    -f, --file <file>               import board from file

    -t, --threads <tc>              use 'tc' many threads

    -h, --headless                  run without gui
```

Original Project/Goals:

The original Hamiltonian Path Problem solver I had in mind for this project is quite different from the one I chose to implement. I planned to use Frank Rubin's algorithm to exhaustively explore graphs for Hamiltonian Cycles, which could be used to solve a path problem by first reducing the problem to a cycle problem, expanding the graph with an extra vertex that connects to all other vertices (undirected edge).

Unfortunately, though Rubin's paper clearly detailed classes of rules to apply to the graph (represented as a tuple of the set of vertices and the set of edges), and conditions to check at each stage to assess whether the current configuration of sets of edges was "admissible", they were hard to implement as the paper focused more abstractly on these than it detailed how to use these in a procedure. Though perhaps an implementation of Rubin's algorithm exists somewhere, I certainly did not find any evidence of this online. This caused some frustration. Though Rubin's algorithm is referenced in many places as "efficient", no runtime analysis was found anywhere. This is not to say that I doubt the efficiency of the algorithm, but had an implementation been found, I'm sure an estimate of the average runtime could might have also been noted by the programmer.

Nevertheless, some parts of Rubin's algorithm did make their way into the solver.

Background:

This project builds off of a previous project I started a couple years ago. Before having had any introduction to AI or computational theory, I became interested in finding a Hamiltonian Path after encountering a puzzle that required me to find one, given a starting point, in a video game. I wanted to implement a solver for one after finding another puzzle like it later in the game. After creating a recursive brute force solver (crazy, right?), I realized just how inefficient that algorithm was. I also noticed some patterns that led the algorithm down paths that could not arrive at a solution. Over time, I tried to find patterns that would help prune the search (although, at the time I did not have the vocabulary to describe much of the phenomena I relearned more formally in my introductory AI course). I implemented some heuristics to detect these patterns and backtrack as necessary. Though these were specific to the grid graphs/puzzles I was considering at the time, some of them apply to more general graphs and were also discovered by Rubin in the 70's. For instance, Rubin's paper defined a failure rule (F1) that checked whether any vertices were isolated/disconnected/unreachable by the rest of the graph. Similarly, I implemented a block detection algorithm to check whether the grid had been divided by the current path. I also wrote a "oneWayOut()" function that checks if any grid square has only one way in/out, similar to rule F3 on Rubin's paper.

Description:

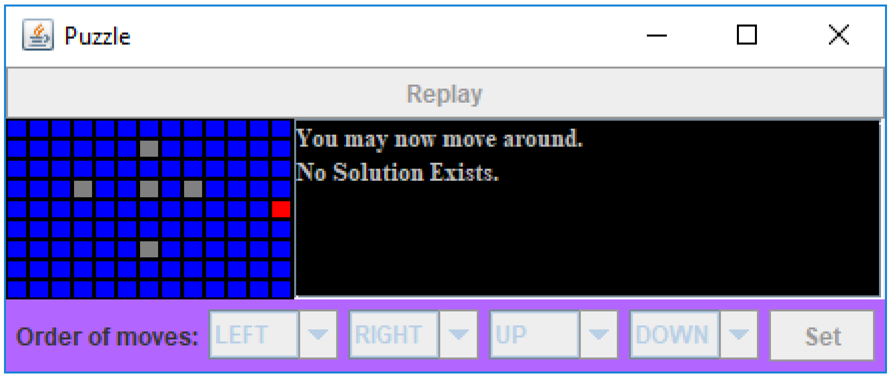
To clarify, this project builds off of a solver that was written before this semester. Though this was mainly a project I developed independent of any assignment/course, I feel it would be dishonest of me to submit this project to be graded as a whole. For this course specifically, I have modified the algorithm to be parallelized by using the Java Executor library. I figured this would be the best way to separate out the tasks with the abstraction of workers. Since it would be impossible to perform different calculations on the same board using synchronization mechanisms, or at least efficiently and intuitively, creating independent subproblems (like what Rubin suggested with his algorithm) seemed like a better approach. First, a fixed thread pool is created, specifying the number of threads/workers available to accomplish the tasks. Then, the space is examined for potential parallelism. The solver does a breadth-first search on the grid to view the possible moves to make, or actions/edges to take. Once a certain level has enough possible moves to use all of the available threads, each task (a different configuration of the board with a partial path) is submitted to the executor to manage and schedule them. The main thread collects the futures of these tasks into a queue to find out when any of them finishes. If the task was successful, the executor performs a shutdown and returns the solution. Otherwise, the future is removed from the queue and the main thread continues to poll these to see if anyone has finished.

In the case that a suitable number of tasks cannot be found for the number of threads, the algorithm detects this and simply runs one task (the initial configuration). If the algorithm couldn't find a big enough level for parallelism, then the search space must have been exhausted and the main thread can just do the job itself. I sadly did not devise a way to make use of a small number of threads instead of one. Though perhaps it would have been better to simply check if any of the terminal-states was a goal state. Sadly I did not have time to do this.

Figures:

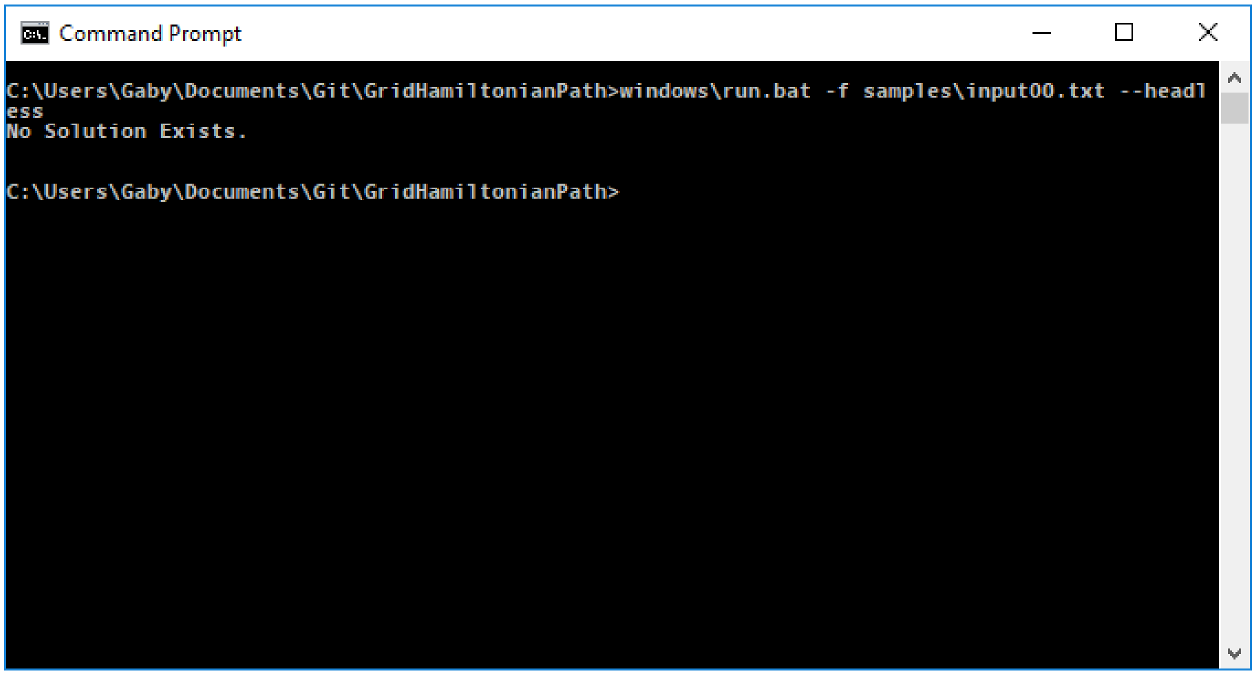
For reference, here is a legend of the different colors on the squares of the grid:

1. Free square: **BLUE** (empty space)
2. Used square: **RED** (part of the current path)
3. Block square: **GRAY** (blocked space; unusable)



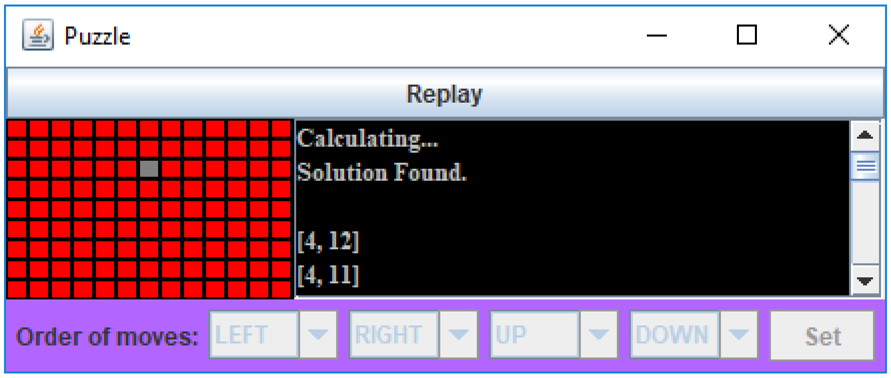
```
You may now move around.
No Solution Exists.
```

Figure 1: *GUI run of input00.txt. An initial admissibility test is performed to check if a solution is possible (using domino tileability, blob detection, and the "oneWayOut" test).*



```
C:\Users\Gaby\Documents\Git\GridHamiltonianPath>windows\run.bat -f samples\input00.txt --headl
ess
No Solution Exists.

C:\Users\Gaby\Documents\Git\GridHamiltonianPath>
```

Figure 2: *TTY run of input00.txt. (See Figure 1).*



```
Calculating...
Solution Found.

[4, 12]
[4, 11]
```

Figure 3*: GUI run of input01.txt. A solution was found and displayed on the textfield.*

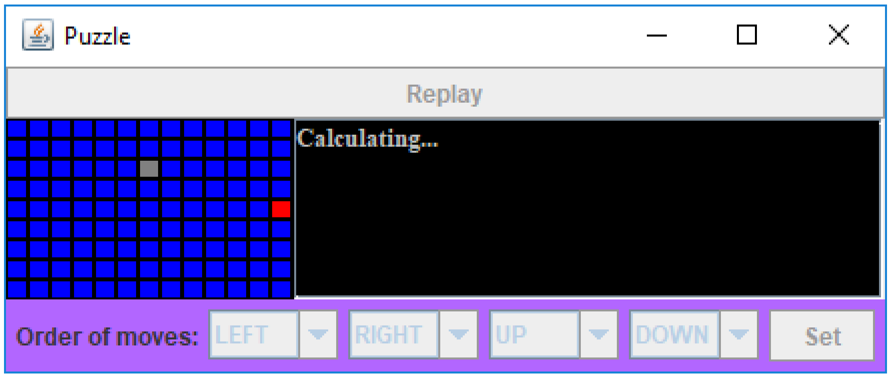
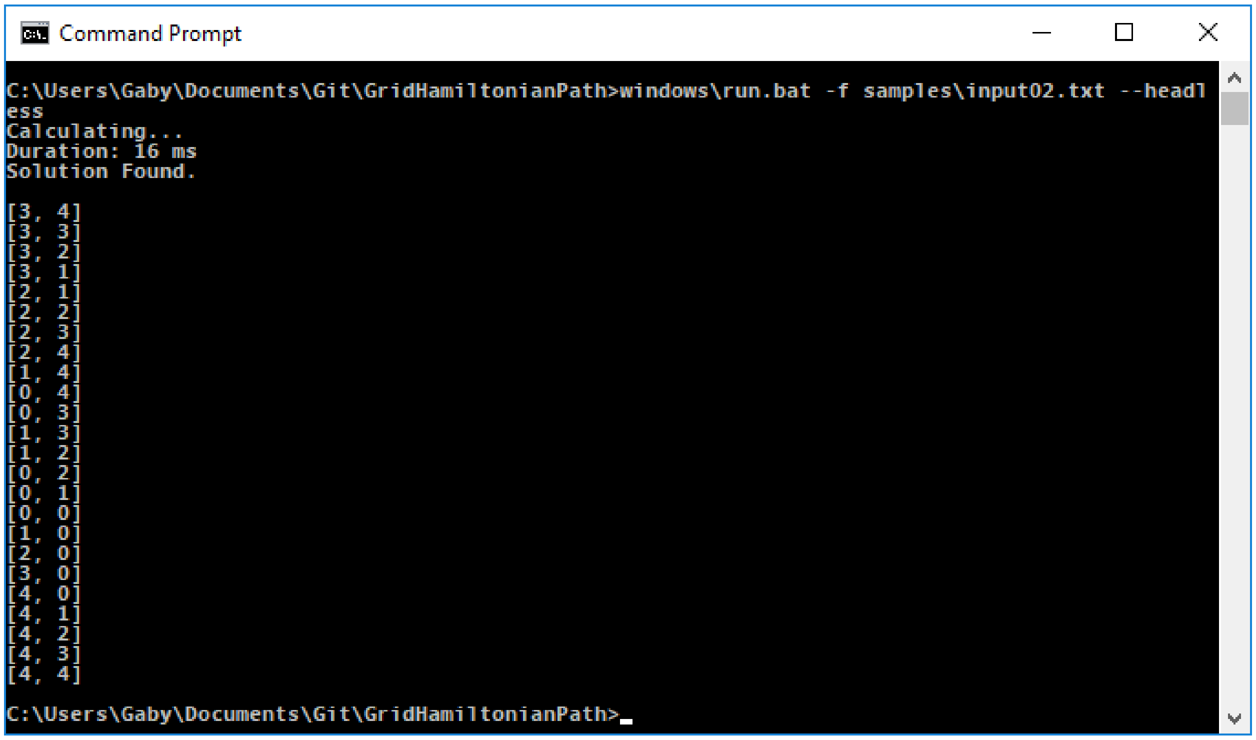
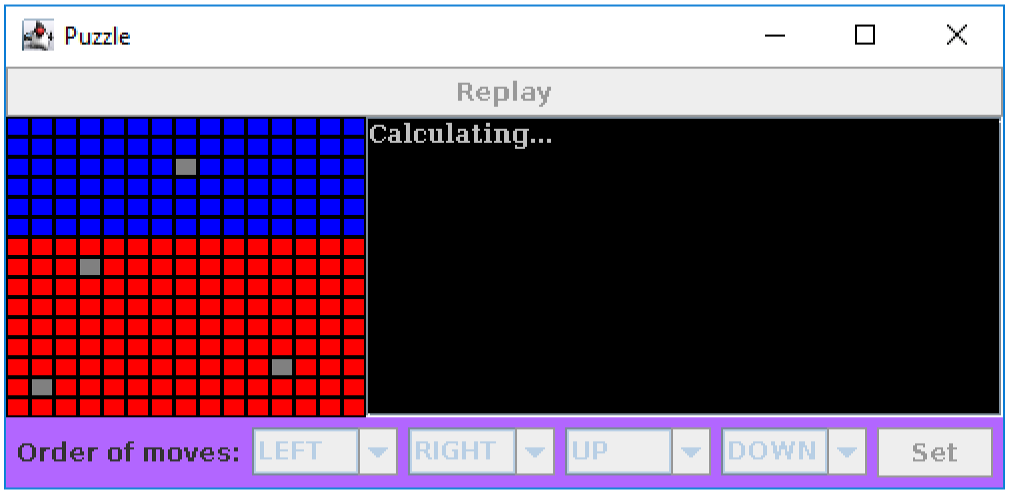


Figure 4: *GUI run of input01.txt. This run is using the original project's sequential algorithm. Without the extra tasks, the algorithm takes a bit of time (though not a lot) to find a solution. However, given a different ordering of moves, a solution can be found much quicker. This demonstrates how the concurrent version of the program benefits by exploring different paths initially.*



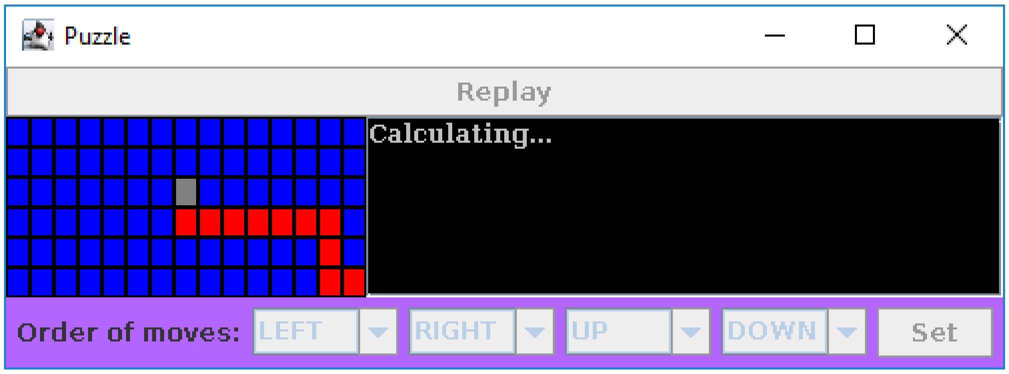
```
C:\Users\Gaby\Documents\Git\GridHamiltonianPath>windows\run.bat -f samples\input02.txt --headl
ess
Calculating...
Duration: 16 ms
Solution Found.

[3, 4]
[3, 3]
[3, 2]
[3, 1]
[2, 1]
[2, 2]
[2, 3]
[2, 4]
[1, 4]
[0, 4]
[0, 3]
[1, 3]
[1, 2]
[0, 2]
[0, 1]
[0, 0]
[1, 0]
[2, 0]
[3, 0]
[4, 0]
[4, 1]
[4, 2]
[4, 3]
[4, 4]

C:\Users\Gaby\Documents\Git\GridHamiltonianPath>_
```

Figure 5: *TTY run of input02.txt. The path and elapsed time are both displayed in the standard output stream.*



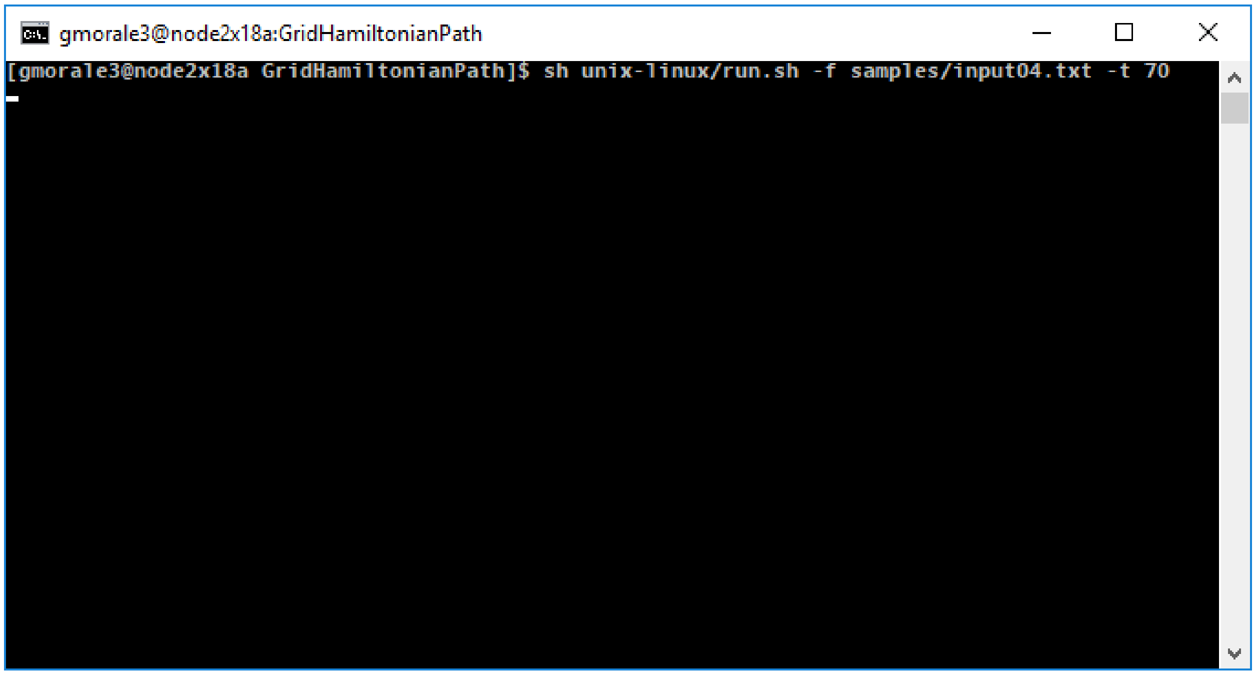
```
Calculating...
```

Figure 6: *GUI run of input03.txt. This board is too big to find a fast solution in a reasonable amount of time. A partial path has been set that covers most of the difficult area of the board. The algorithm can then be commenced to see if a full path exists using the current path. Sadly, one does not exist.*



```
Calculating...
```

Figure 7: *GUI run of input04.txt. This input file was created merely to show that solving this board would be equivalent to solving the previous one at its present configuration (ignore the path on the image).*



```
[gmorale3@node2x18a GridHamiltonianPath]$ sh unix-linux/run.sh -f samples/input04.txt -t 70
```

Figure 8: *Terminal view of GUI run of input04.txt form Figure 7. Here, 70 threads on node2x18a on the CSUG network were used to do the calculation. Ultimately, no solution was found and I no runtime calculations are performed on GUI runs of the program (not pictured above).*

Results:

When solutions exist, and the input sizes are small enough, generally the solver is effective at finding these solutions in a reasonable amount of time. Some solutions can be found more quickly using a different ordering of moves. The concurrent version of the algorithm somewhat improves on the original algorithm in this respect since it runs tasks with different starting paths. When solutions do not exist, the solver takes exponential time to figure that out. Given that the graphs are grids, a vertex has at most 3 undirected edges connected to it. Thus, the brute-force runtime would take $O(3^n)$ time. The different heuristics significantly prune bad paths, and some rules can even determine whether a board is unsolvable from the initial state (mainly the domino tileability algorithm). However, these can reduce the runtime by at most a small exponential, i.e. $O(3^n) – O(3^k) = O(3^n)$ for any pruned paths at level $n – k – 1$.

Sadly, I did not use significantly large boards to collect sufficient runtime data here. The ones that were large enough would have taken a significant amount of time to run.

Future Work:

There are many ambitious I had for this project that were not implemented for the sake of time. I intended to find ways of including more of Rubin's rules/tests at each step of the algorithm to prune the search space some more. As of yet, I am not sure how I would have done this. Additionally, the parallelization potential calculations were not as great as I would have hoped. I wanted to find a way to use the Java Executor libraries to create tasks recursively and dynamically and submit them to the thread pool. The load would have been balanced better between threads in this way if tasks could determine whether other tasks had finished and threads were currently available, in which case, they could split up their work and submit subtasks to the pool.